07｜构建模式：Go Module的6类常规操作

2021-10-27 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 14:09 大小 12.97M

你好，我是 Tony Bai。

通过上一节课的讲解，我们掌握了 Go Module 构建模式的基本概念和工作原理，也初步学会了如何通过 go mod 命令，将一个 Go 项目转变为一个 Go Module，并通过 Go Module 构建模式进行构建。

但是，围绕一个 Go Module，Go 开发人员每天要执行很多 Go 命令对其进行维护。这些维护又是怎么进行的呢？

具体来说，维护 Go Module 无非就是对 Go Module 依赖包的管理。但在具体工作中还有很多情况，我们接下来会拆分成六个场景，层层深入给你分析。可以说，学好这些是每个 Go 开发人员成长的必经之路。

我们首先来看一下日常进行 Go 应用开发时遇到的最为频繁的一个场景：**为当前项目添加一个依赖包。**

为当前 module 添加一个依赖

在一个项目的初始阶段，我们会经常为项目引入第三方包，并借助这些包完成特定功能。即便是项目进入了稳定阶段，随着项目的演进，我们偶尔还需要在代码中引入新的第三方包。

那么我们如何为一个 Go Module 添加一个新的依赖包呢？

我们还是以上一节课中讲过的 module-mode 项目为例。如果我们要为这个项目增加一个新依赖：github.com/google/uuid，那需要怎么做呢？

我们首先会更新源码，就像下面代码中这样：

复制代码

```
package main

import (
  "github.com/google/uuid"
  "github.com/sirupsen/logrus"
)

func main() {
  logrus.Println("hello, go module mode")
  logrus.Println(uuid.NewString())
}
```

新源码中，我们通过 import 语句导入了 github.com/google/uuid，并在 main 函数中调用了 uuid 包的函数 NewString。此时，如果我们直接构建这个 module，我们会得到一个错误提示：

复制代码

```
$go build
main.go:4:2: no required module provides package github.com/google/uuid; to ad
  go get github.com/google/uuid
```

Go 编译器提示我们，go.mod 里的 require 段中，没有哪个 module 提供了 github.com/google/uuid 包，如果我们要增加这个依赖，可以手动执行 go get 命令。那我们就来按照提示手工执行一下这个命令：

复制代码

```
$go get github.com/google/uuid
go: downloading github.com/google/uuid v1.3.0
go get: added github.com/google/uuid v1.3.0
```

你会发现，go get 命令将我们新增的依赖包下载到了本地 module 缓存里，并在 go.mod 文件的 require 段中新增了一行内容：

复制代码

```
require (
  github.com/google/uuid v1.3.0 //新增的依赖
  github.com/sirupsen/logrus v1.8.1
)
```

这新增的一行表明，我们当前项目依赖的是 uuid 的 v1.3.0 版本。我们也可以使用 go mod tidy 命令，在执行构建前自动分析源码中的依赖变化，识别新增依赖项并下载它们：

复制代码

```
$go mod tidy
go: finding module for package github.com/google/uuid
go: found github.com/google/uuid in github.com/google/uuid v1.3.0
```

对于我们这个例子而言，手工执行 go get 新增依赖项，和执行 go mod tidy 自动分析和下载依赖项的最终效果，是等价的。但对于复杂的项目变更而言，逐一手工添加依赖项显然很没有效率，go mod tidy 是更佳的选择。

到这里，我们已经了解了怎么为当前的 module 添加一个新的依赖。但是在日常开发场景中，我们需要对依赖的版本进行更改。那这又要怎么做呢？下面我们就来看看下面升、降级修改依赖版本的场景。

升级 / 降级依赖的版本

我们先以对依赖的版本进行降级为例，分析一下。

在实际开发工作中，如果我们认为 Go 命令自动帮我们确定的某个依赖的版本存在一些问题，比如，引入了不必要复杂性导致可靠性下降、性能回退等等，我们可以手工将它降级为之前发布的某个兼容版本。

那这个操作依赖于什么原理呢？

答案就是我们上一节课讲过“语义导入版本”机制。我们再来简单复习一下，Go Module 的版本号采用了语义版本规范，也就是版本号使用 vX.Y.Z 的格式。其中 X 是主版本号，Y 为次版本号 (minor)，Z 为补丁版本号 (patch)。主版本号相同的两个版本，较新的版本是兼容旧版本的。如果主版本号不同，那么两个版本是不兼容的。

有了语义版本号作为基础和前提，我们就可以从容地手工对依赖的版本进行升降级了，Go 命令也可以根据版本兼容性，自动选择出合适的依赖版本了。

我们还是以上面提到过的 logrus 为例，logrus 现在就存在着多个发布版本，我们可以通过下面命令来进行查询：

复制代码

```
$go list -m -versions github.com/sirupsen/logrus
github.com/sirupsen/logrus v0.1.0 v0.1.1 v0.2.0 v0.3.0 v0.4.0 v0.4.1 v0.5.0 v0
```

在这个例子中，基于初始状态执行的 go mod tidy 命令，帮我们选择了 logrus 的最新发布版本 v1.8.1。如果你觉得这个版本存在某些问题，想将 logrus 版本降至某个之前发布的兼容版本，比如 v1.7.0，**那么我们可以在项目的 module 根目录下，执行带有版本号的 go get 命令：**

复制代码

```
$go get github.com/sirupsen/logrus@v1.7.0
go: downloading github.com/sirupsen/logrus v1.7.0
go get: downgraded github.com/sirupsen/logrus v1.8.1 => v1.7.0
```

从这个执行输出的结果，我们可以看到，go get 命令下载了 logrus v1.7.0 版本，并将 go.mod 中对 logrus 的依赖版本从 v1.8.1 降至 v1.7.0。

当然我们也可以使用万能命令 go mod tidy 来帮助我们降级，但前提是首先要用 go mod edit 命令，明确告知我们要依赖 v1.7.0 版本，而不是 v1.8.1，这个执行步骤是这样的：

```
$go mod edit -require=github.com/sirupsen/logrus@v1.7.0
$go mod tidy
go: downloading github.com/sirupsen/logrus v1.7.0
```

降级后，我们再假设 logrus v1.7.1 版本是一个安全补丁升级，修复了一个严重的安全漏洞，而且我们必须使用这个安全补丁版本，这就意味着我们需要将 logrus 依赖从 v1.7.0 升级到 v1.7.1。

我们可以使用与降级同样的步骤来完成升级，这里我只列出了使用 go get 实现依赖版本升级的命令和输出结果，你自己动手试一下。

```
$go get github.com/sirupsen/logrus@v1.7.1
go: downloading github.com/sirupsen/logrus v1.7.1
go get: upgraded github.com/sirupsen/logrus v1.7.0 => v1.7.1
```

好了，到这里你就学会了如何对项目依赖包的版本进行升降级了。

但是你可能会发现一个问题，在前面的例子中，Go Module 的依赖的主版本号都是 1。根据我们上节课中学习的语义导入版本的规范，在 Go Module 构建模式下，当依赖的主版本号为 0 或 1 的时候，我们在 Go 源码中导入依赖包，不需要在包的导入路径上增加版本号，也就是：

```
import github.com/user/repo/v0 等价于 import github.com/user/repo
import github.com/user/repo/v1 等价于 import github.com/user/repo
```

但是，如果我们要依赖的 module 的主版本号大于 1，这又要怎么办呢？接着我们就来看看这个场景下该如何去做。

添加一个主版本号大于 1 的依赖

这里，我们还是先来回顾一下，上节课我们讲的语义版本规则中对主版本号大于 1 情况有没有相应的说明。

有的。语义导入版本机制有一个原则：**如果新旧版本的包使用相同的导入路径，那么新包与旧包是兼容的**。也就是说，如果新旧两个包不兼容，那么我们就应该采用不同的导入路径。

按照语义版本规范，如果我们要为项目引入主版本号大于 1 的依赖，比如 v2.0.0，那么由于这个版本与 v1、v0 开头的包版本都不兼容，我们在导入 v2.0.0 包时，不能再直接使用 github.com/user/repo，而要使用像下面代码中那样不同的包导入路径：

复制代码

```
import github.com/user/repo/v2/xxx
```

也就是说，如果我们要为 Go 项目添加主版本号大于 1 的依赖，我们就需要使用“语义导入版本”机制，**在声明它的导入路径的基础上，加上版本号信息**。我们以“向 module-mode 项目添加 github.com/go-redis/redis 依赖包的 v7 版本”为例，看看添加步骤。

首先，我们在源码中，以空导入的方式导入 v7 版本的 github.com/go-redis/redis 包：

复制代码

```
package main

import (
  _ "github.com/go-redis/redis/v7"
  "github.com/google/uuid"
  "github.com/sirupsen/logrus"
)

func main() {
  logrus.Println("hello, go module mode")
  logrus.Println(uuid.NewString())
}
```

接下来的步骤就与添加兼容依赖一样，我们通过 go get 获取 redis 的 v7 版本：

```
$go get github.com/go-redis/redis/v7
go: downloading github.com/go-redis/redis/v7 v7.4.1
go: downloading github.com/go-redis/redis v6.15.9+incompatible
go get: added github.com/go-redis/redis/v7 v7.4.1
```

我们可以看到，go get 为我们选择了 go-redis v7 版本下当前的最新版本 v7.4.1。

不过呢，这里说的是为项目添加一个主版本号大于 1 的依赖的步骤。有些时候，出于要使用依赖包最新功能特性等原因，我们可能需要将某个依赖的版本升级为其不兼容版本，也就是主版本号不同的版本，这又该怎么做呢？

我们还以 go-redis/redis 这个依赖为例，将这个依赖从 v7 版本升级到最新的 v8 版本看看。

升级依赖版本到一个不兼容版本

我们前面说了，按照语义导入版本的原则，不同主版本的包的导入路径是不同的。所以，同样地，我们这里也需要先将代码中 redis 包导入路径中的版本号改为 v8：

```
import (
  _ "github.com/go-redis/redis/v8"
  "github.com/google/uuid"
  "github.com/sirupsen/logrus"
)
```

接下来，我们再通过 go get 来获取 v8 版本的依赖包：

```
$go get github.com/go-redis/redis/v8
go: downloading github.com/go-redis/redis/v8 v8.11.1
go: downloading github.com/dgryski/go-rendezvous v0.0.0-20200823014737-9f7001d
go: downloading github.com/cespare/xxhash/v2 v2.1.1
go get: added github.com/go-redis/redis/v8 v8.11.1
```

这样，我们就完成了向一个不兼容依赖版本的升级。是不是很简单啊！

但是项目继续演化到一个阶段的时候，我们可能还需要移除对之前某个包的依赖。

移除一个依赖

我们还是看前面 go-redis/redis 示例，如果我们这个时候不需要再依赖 go-redis/redis 了，你会怎么做呢？

你可能会删除掉代码中对 redis 的空导入这一行，之后再利用 go build 命令成功地构建这个项目。

但你会发现，与添加一个依赖时 Go 命令给出友好提示不同，这次 go build 没有给出任何关于项目已经将 go-redis/redis 删除的提示，并且 go.mod 里 require 段中的 go-redis/redis/v8 的依赖依旧存在着。

我们再通过 go list 命令列出当前 module 的所有依赖，你也会发现 go-redis/redis/v8 仍出现在结果中：

复制代码

```
$go list -m all
github.com/bigwhite/module-mode
github.com/cespare/xxhash/v2 v2.1.1
github.com/davecgh/go-spew v1.1.1
... ...
github.com/go-redis/redis/v8 v8.11.1
... ...
gopkg.in/yaml.v2 v2.3.0
```

这是怎么回事呢？

其实，要想彻底从项目中移除 go.mod 中的依赖项，仅从源码中删除对依赖项的导入语句还不够。这是因为如果源码满足成功构建的条件，go build 命令是不会“多管闲事”地清理 go.mod 中多余的依赖项的。

那正确的做法是怎样的呢？我们还得用 go mod tidy 命令，将这个依赖项彻底从 Go Module 构建上下文中清除掉。go mod tidy 会自动分析源码依赖，而且将不再使用的依赖从 go.mod 和 go.sum 中移除。

到这里，其实我们已经分析了 Go Module 依赖包管理的 5 个常见情况了，但其实还有一种特殊情况，需要我们借用 vendor 机制。

特殊情况：使用 vendor

你可能会感到有点奇怪，为什么 Go Module 的维护，还有要用 vendor 的情况？

其实，vendor 机制虽然诞生于 GOPATH 构建模式主导的年代，但在 Go Module 构建模式下，它依旧被保留了下来，并且成为了 Go Module 构建机制的一个很好的补充。特别是在一些不方便访问外部网络，并且对 Go 应用构建性能敏感的环境，比如在一些内部的持续集成或持续交付环境 (CI/CD) 中，使用 vendor 机制可以实现与 Go Module 等价的构建。

和 GOPATH 构建模式不同，Go Module 构建模式下，我们再也无需手动维护 vendor 目录下的依赖包了，Go 提供了可以快速建立和更新 vendor 的命令，我们还是以前面的 module-mode 项目为例，通过下面命令为该项目建立 vendor：

复制代码

```
$go mod vendor
$tree -LF 2 vendor
vendor
├── github.com/
│   ├── google/
│   ├── magefile/
│   └── sirupsen/
├── golang.org/
│   └── x/
└── modules.txt
```

我们看到，go mod vendor 命令在 vendor 目录下，创建了一份这个项目的依赖包的副本，并且通过 vendor/modules.txt 记录了 vendor 下的 module 以及版本。

如果我们要基于 vendor 构建，而不是基于本地缓存的 Go Module 构建，我们需要在 go build 后面加上 -mod=vendor 参数。

在 Go 1.14 及以后版本中，如果 Go 项目的顶层目录下存在 vendor 目录，那么 go build 默认也会优先基于 vendor 构建，除非你给 go build 传入 -mod=mod 的参数。

小结

好了，到这里，我们就完成了维护 Go Module 的全部常见场景的学习了，现在我们一起来回顾一下吧。

在通过 go mod init 为当前 Go 项目创建一个新的 module 后，随着项目的演进，我们在日常开发过程中，会遇到多种常见的维护 Go Module 的场景。

其中最常见的就是为项目添加一个依赖包，我们可以通过 go get 命令手工获取该依赖包的特定版本，更好的方法是通过 go mod tidy 命令让 Go 命令自动去分析新依赖并决定使用新依赖的哪个版本。

另外，还有几个场景需要你记住：

- 通过 go get 我们可以升级或降级某依赖的版本，如果升级或降级前后的版本不兼容，这里千万注意别忘了变化包导入路径中的版本号，这是 Go 语义导入版本机制的要求；
- 通过 go mod tidy，我们可以自动分析 Go 源码的依赖变更，包括依赖的新增、版本变更以及删除，并更新 go.mod 中的依赖信息。
- 通过 go mod vendor，我们依旧可以支持 vendor 机制，并且可以对 vendor 目录下缓存的依赖包进行自动管理。

在了解了如何应对 Go Modules 维护的日常工作场景后，你是不是有一种再也不担心 Go 源码构建问题的感觉了呢？

思考题

如果你是一个公共 Go 包的作者，在发布你的 Go 包时，有哪些需要注意的地方？

感谢你和我一起学习，也欢迎你把这节课分享给更多对 Go 构建模式感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 06 | 构建模式：Go是怎么解决包依赖管理问题的？

精选留言 (16)

写留言

Aaron Liu
2021-10-27

如果之前引用的包是v1，之后升级v2，go get可以替换引用的包，但源码里的import要怎么改，如果很多go文件都引用了呢

展开

2 4

lesserror

2021-10-27

Tony Bai 老师这一讲的内容很实用，可以说有很多Go教程都没有涉及到这块知识的归纳总结。
麻烦老师抽空回答一下我以下的疑问：

1. 空导入的方式的作用吗？我看很多源码中有使用这种包导入的方式。...

展开

2 1

return

2021-10-27

老师讲太好了， 有主线 有关键细节，
请教老师， 关于vendor， 存好副本后， 一般在其他地方怎么用呢，
手动传输过去 还是 上传到代码库再下载呢

1 1

一步

2021-10-27

go 的命令好多，老师来讲讲 使用 Makefile 来管理的方法

1

进化菌

2021-10-28

感觉go mod tidy很常用，通过例子来学习的感觉也挺好的。
还有，vendor保留下来有它的道理吧，每个项目就应该管自己的依赖~

张诚

2021-10-28

讲了很多不知道的知识点，非常体系化的学校内容，棒。

是谁偷了我的快乐

2021-10-28

go mod edit -require=github.com/sirupsen/logrus@v1.7.0这个指令在win 上的golangd好像会因github 后面的那个．识别不出来path,加引号变成 go mod edit -require="github.com/sirupsen/logrus@v1.7.0"就可以了

展开

付帅帅

2021-10-28

$ go list -m -versions github.com/sirupsen/logrus
github.com/sirupsen/logrus

为什么我执行后只有一个包名 没有出现版本号

jacky

2021-10-28

讲得挺好，就是更新有点慢啊，这得更到啥时候

展开

暴风雪

2021-10-28

这个模式跟php的composer很像

展开

不负青春不负己

2021-10-27

个人理解 ：1 .代码使用 Go module 构建包含 go.mod 和go.sum 解决依赖，
2. 项目结构符合 社区标准
3. 涉及本身代码license以及其他第三方包的 license

...

展开

Bynow

2021-10-27

invalid version: module contains a go.mod file, so major version must be compatible: should be v0 or v1, not v3 最近下载自己上传的go包 经常会有这个问题，应该是语义版本没理解透彻

Bynow

2021-10-27

公共go包的作者应该比较关注 包的tag 和 go mod init 后面的v(版本)吧。

罗杰 神

2021-10-27

虽然已用 Go 开发三年了，老师把所有的坑点都说了，还顺便讲了一些我不知道🙈的知识点，受益匪浅！

展开

Geek_eb6c14

2021-10-27

不能出个视频吗？

展开

2

aoe 酥

2021-10-27

知道了 go mod tidy 命令再也不用因依赖问题瑟瑟发抖了